Towards Point-Based Acquisition and Rendering of Large Real-World Environments

WAGNER T. CORRÊA[1], SHACHAR FLEISHMAN[2], CLÁUDIO T. SILVA[3]

[1]Princeton University, 35 Olden St., Princeton, NJ 08540, USA
wtcorrea@cs.princeton.edu
[2]Tel-Aviv University, Schriber Building, Tel Aviv, 69978, Israel
shacharf@math.tau.ac.il
[3]OGI School of Science and Technology, 20000 NW Walker Road, Beaverton, OR, 97006, USA
Work performed while at AT&T Labs-Research
csilva@cse.ogi.edu

Abstract. This paper describes a pipeline for the acquisition and rendering of large real-world environments. In the acquisition phase, we use a laser rangefinder to capture the geometry of an environment, and a digital camera to capture its colors. In the rendering phase, we use a cluster of commodity PCs to render high-resolution images of the environment at interactive frame rates. In this paper, we describe in detail our scanning hardware, the tools we use to minimize the acquisition artifacts in the 3D scans, the procedure to register the scans to each other, and how to map colors from a photograph to a scan. We also present a sequential, out-of-core rendering approach that uses multiple threads to overlap rendering, visibility computation, and disk operations. Finally, we show how to use the sequential rendering approach as a building block for a parallel rendering system that uses a cluster of PCs to drive a high-resolution, multi-projector display wall. Our acquisition approach allows us to capture environments that would be extremely difficult to model by hand, and our rendering approach allows us to use inexpensive PCs, instead of high-end graphics workstations, to visualize those environments at interactive frame rates.

1 Introduction

Interactive rendering of realistic environments has been a focus of computer graphics research for many years. Traditionally, researchers have modeled the geometric and photometric properties of an environment manually, and the resulting models have been polygonal soups or meshes. Recently, 3D scanning technology has allowed researchers to capture those properties directly from real-world environments [4, 19, 22, 33], and the use of points instead of polygons as rendering primitives has become widespread [12, 23, 26]. Most of the research on acquiring and rendering point-based models has focused on single objects [4, 19]. In contrast, we share the goals of Nyland et al. [22], and focus our work on large real-world environments.

In this paper, we describe a complete pipeline for acquiring and rendering a point-based 3D model of a large real-world environment (Figure 1). To acquire the geometry of an environment, we use a time-of-flight laser rangefinder which gives us a dense point cloud. To obtain the colors of the environment, we use a high-resolution digital camera, and then map the colors in the photographs onto the scanned points. Finally, to render high-resolution images of the model at interactive frame rates, we use a cluster of commodity PCs to drive a multi-projector display wall.

This paper describes each step of this pipeline in detail, discussing the challenges involved and how we address them. We start with a brief review of related work in Section 2. In Section 3 we describe our scanning hardware, and the tools we have developed to analyze and process the 3D scans to identify and minimize acquisition artifacts such as noise and bias. Because of occlusion, we need several scans of the same area to obtain a complete model. In Section 4 we address the problems of merging the scans into a common coordinate-system and incrementally creating an out-of-core hierarchical representation of the model. Section 5 describes the procedure to capture the imagery and map it onto the geometry. In Section 6 we present our sequential, out-of-core, multi-threaded approach to render the resulting 3D model. Section 7 shows how to use the sequential rendering approach as a building block for a parallel system that employs a cluster of PCs to render high-resolution images of the model on a display wall. In Section 8 we present the results of acquiring and rendering a large and complex cluttered office environment. Finally, in Section 9 we conclude and discuss directions for future work.

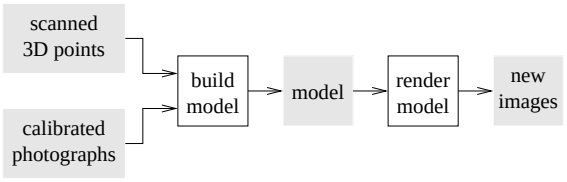


Figure 1: Our acquisition and rendering pipeline.

2 Related Work

Advances in scanning technology have allowed researchers to acquire models with millions of polygons [4, 19]. When rendering such large models, many triangles will have a projected area smaller than a pixel. In this case, it makes sense to render point samples instead of triangles. Recently, many researchers have developed point-based rendering systems [12, 23, 26, 31], but point rendering is in fact quite an old concept. Csuri [9] suggested the idea of using points as primitives to render 3D surfaces more than two decades ago. Levoy and Whitted [20] used points to render differentiable surfaces. Points have also been used to model fuzzy objects such as clouds, fire, and plants [5, 25, 32].

Most model acquisition systems focus on capturing a single object such as a statue [4, 19]. Instead, we are interested in capturing entire environments [22, 33]. The system that is most similar to ours is the one of Nyland et al. [22], but there are several differences between our approaches. Their best procedure to register two scans is manual, while we use a semi-automatic procedure. Their procedure for aligning geometry and imagery is based on automatic edge detection, while we (again) use a semi-automatic procedure. Our procedure requires more work from the user, but is less sensitive to noise in the imagery. Finally, they use a high-end graphics workstation for rendering, and assume the entire model fits in memory. We have developed out-of-core techniques to build, process, and render the model so that we can use inexpensive PCs.

Funkhouser et al. [11] were the first to publish a system that supported models larger than main memory, and performed speculative prefetching. Their approach relied on the from-region visibility algorithm of Teller et al. [34], which requires long preprocessing times, and assumes that the models are made of axis-aligned cells. Our approach is based on the from-point visibility algorithm of Klosowski and Silva [16], which requires very little preprocessing, and makes no assumptions about the geometry of the model.

Aliaga et al. [3] have developed the massive model rendering (MMR) system. MMR employs a large number of acceleration techniques, including replacing distant geometry with image impostors, managing levels of detail, and culling occluded geometry. MMR was perhaps the first published system to handle models with tens of millions of polygons at interactive frame rates. On the other hand, MMR required up to weeks of preprocessing time and expensive high-end graphics workstations. Our approach requires only minutes of preprocessing, and works on a cluster of commodity PCs.

Samanta et al. [28, 29] have developed a sort-first rendering system using a network of PCs. The main focus of their work was on load balancing the geometry processing and rasterization work done on each of the PCs, while we focus on handling very large models.

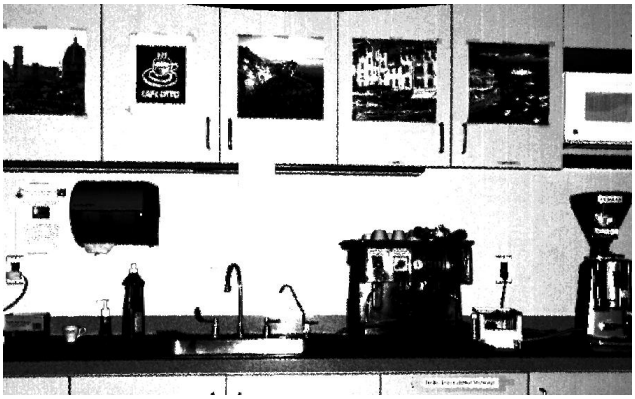
3 Acquiring the Geometry

Figure 2: Our scanner.

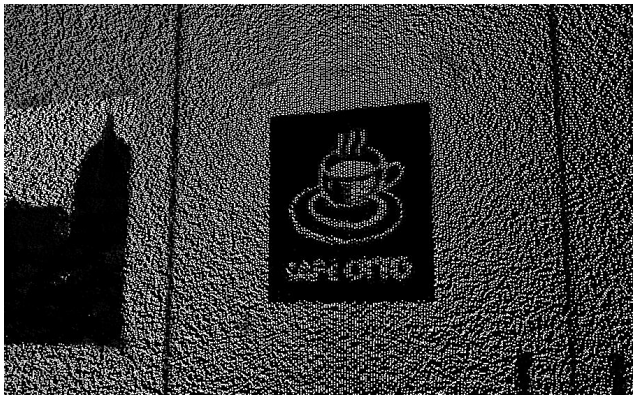
To acquire the geometry of an environment, we use a time-of-flight scanner (Figure 2) called DeltaSphere-3000 [1]. This scanner is small (14 in. by 12 in. by 4 in.), and weighs about 25 lbs. It has an effective range of 40 ft. and an accuracy of about 1/2 in. To acquire a set of point samples, we first place the scanner on top of a tripod at some point inside the environment. We then use a program that communicates with the scanner via wireless network to set the desired field of view and sample density, and start the scanning process. The scanner rotates about its vertical axis, and for each position it acquires a vertical strip of sample points. The result if a set of points, each consisting of its spherical coordinates (r, θ, ϕ) and the intensity i of the energy returned from that point. We thus use the term "*rtpi* sample" to refer to a point sample. Figure 3 shows a sample scan. To acquire that scan, we set the horizontal field of view to 120 degrees, the vertical field of view to 50 degrees, and the sampling density to "low." The scanning process took about 1 minute, and returned 1.1 million points (19 MB) organized as 1604 vertical strips.

The raw data provided by the scanner contains several artifacts, including a large amount of noise, a large difference in resolution between near and far surfaces, and a significant amount of distortion due to intensity bias. In particular, points with too low or too high intensity tend to have unreliable radius. To deal with these problems, we have developed a set of tools to analyze and process the raw scans. One of these tools is a program called *rtpi-histogram*, which plots a histogram of the intensities of the sample points. This allows us to quickly identify a range of reliable intensities, and then use another tool, *rtpi-cut*, to eliminate points outside that range. To equalize the intensity histogram of the remaining points, we use the tool called *rtpi-equalize*.

To minimize the intensity bias of the scanner, we use a calibration image that changes linearly from black to white, and is surrounded by a white background (Figure 4a). We place the image on a flat surface and scan it. From the scan, we build a bias correction table that is used for correcting subsequent scans. Figures 4b and 4d show the scanned calibration image in 3D. We assume that the noise distribution in the scan is of zero mean. The bias in the scan is in the r part of an rtpi sample, not in ϕ or θ. We consider the background to be the ground truth, and we fit the best (in the least squares sense) plane to the background points. A bias is computed for every intensity value $i \in [0, 255]$ as the average height $\overline{H}_i$ (relative to the plane) of the sampled points with intensity i. To minimize the bias, we iterate over the

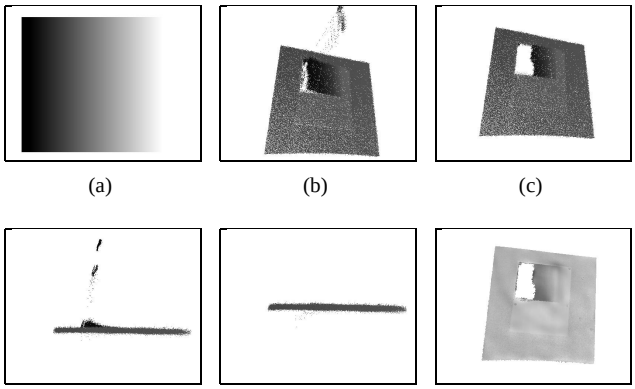


(a) Cafe Otto's kitchen



(b) Closeup view of a cupboard door

Figure 3: A sample scan.



(a) (b) (c)

(d) (e) (f)

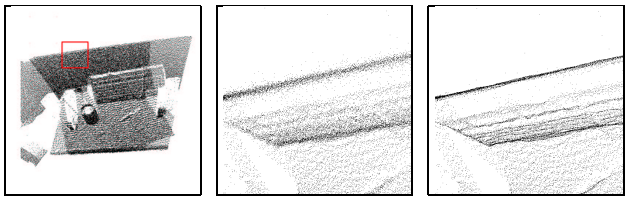
Figure 4: Removing intensity bias. (a) Test pattern. (b) Before removing bias. (c) After removing bias. (d) Side view before. (e) Side view after. (f) Reconstructed surface.

points in the scan, and correct the radius r of every point by computing $r' = r - \overline{H}_i$. Figures 4c and 4e show the result of applying the bias minimization procedure to the calibration image. We casually place the scanner in front of the calibration image, and do not require any precise measurement of the distance or angle between the scanner and calibration image. To verify that the process works, we scanned the calibration image several times from different angles and distances, and applied the tool to these new scans. We obtained very similar correction tables for all cases.

To remove noise, we apply the *moving least squares (MLS) projection* of Levin [2, 18], with a weight function

$$\theta(d) = e^{-\frac{d^2}{h^2}}.$$

We set $h = 4a$, where a is the scanner's accuracy (0.5 in.). Figure 4f shows a smoothed version of the bias calibration image, and Figure 5 shows the application of the noise removal procedure to a scanned desktop.



(a) (b) (c)

Figure 5: Removing noise. (a) Scan of a desktop. (b) Before removing noise. (c) After removing noise.

4 Merging the Geometry

A single scan will have holes due to occlusion, and will sample near and far objects at different resolutions. Thus, to get a more complete model, or to obtain a more uniform resolution, we need to scan the environment from several different locations. Automatically determining the best set of locations for scanning is a hard problem [10, 24]. We simply select the scanning locations ourselves, trying to minimize the number of scans necessary for a good coverage of the environment, and making sure that there is some overlap between the scans.

After we have a set of scans, we need to align them in a common coordinate system. To do that, we have developed a tool called *rtpi-register*. Currently, *rtpi-register* only performs pairwise scan alignment. For a given pair of scans, the user selects a few corresponding points on each scan, and then the computer finds the best rigid transformation (in the least squares sense) that aligns the two scans using Horn's technique [14].

Finally, after aligning the scans, we have to build a hierarchical representation of the combined model, which will be use later for rendering. Since a high-resolution scan is about 100 MB large, the final size of a combined model is in the order of gigabytes. Since one of our goals is to process and render these models using commodity PCs with

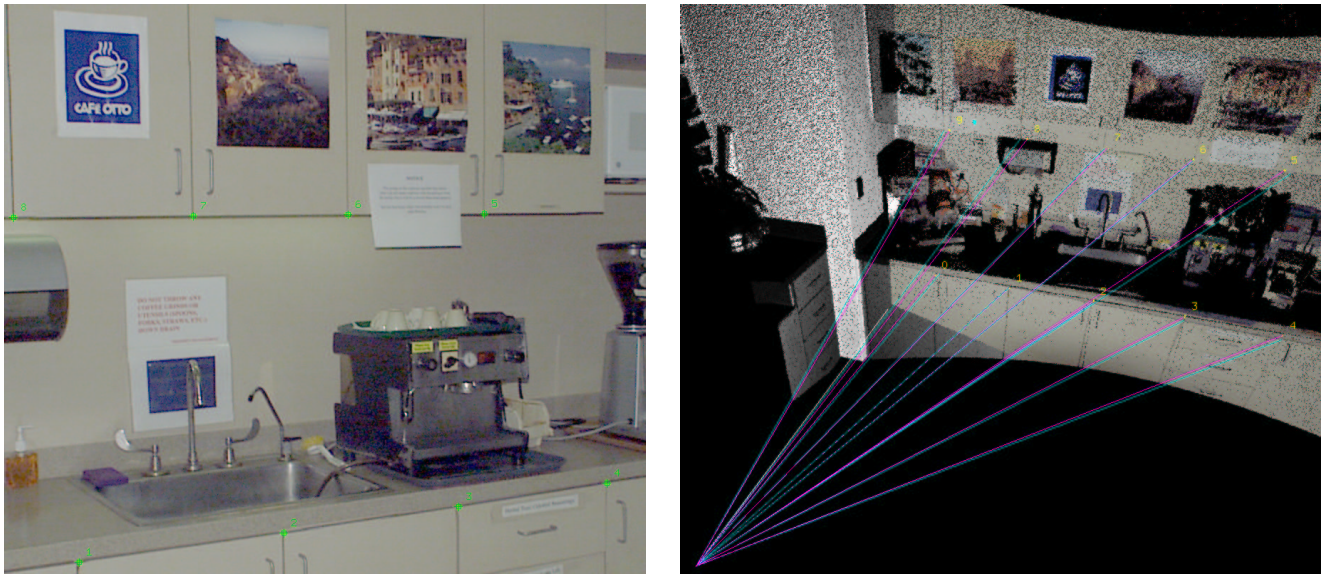


Figure 6: The semi-automatic imagery-geometry registration process.

small memory, we need an *out-of-core* algorithm to build the hierarchical representation of a given model. Moreover, because we may want to update the model with more scans to fill in holes, we need an *incremental* algorithm. We have developed such an algorithm [7]. In practice, a tool called *oct-gen* generates an octree [30] with the contents of a given scan, and a tool called *oct-add* uses our out-of-core, incremental algorithm to add a scan to an existing octree.

5 Acquiring the Imagery

The 3D scanner only provides us with geometric information. To capture the colors of the environment, we take photographs using a Kodak DCS-300 digital camera. The first step in acquiring the imagery is calibrating the *intrinsic* parameters of the camera. We use Willson's freeware implementation [36] of Tsai's camera calibration algorithm [35]. Tsai's camera model has five intrinsic parameters, namely the focal length, the first-order coefficient of radial distortion, the coordinates of the image center, and a scale factor. To calibrate these intrinsic parameters, we first take a photograph of a planar checkerboard pattern with known geometry. Then, we find the image location of the checkerboard corners with subpixel accuracy using Bouguet's camera calibration toolbox [6]. Finally, we pass the 3D and 2D locations of the checkerboard corners to *ccal_fo*, Willson's program for finding the intrinsic camera parameters using full non-linear optimization given coplanar calibration data.

After calibrating the intrinsic camera parameters, we take photographs of the environment. We keep the same camera settings for all photographs to avoid having to recalibrate the intrinsic parameters. For each photograph, we first remove its radial distortion, using a warp based on the coefficient found above, and then find the *extrinsic* camera parameters, namely the position (translation) and orientation (rotation) of the camera when we took the photograph relative to an arbitrary global coordinate system. To find the extrinsic camera parameters, we use an interactive program to specify corresponding 2D points on the undistorted photograph and 3D points on a scan. These corresponding points are then passed along with the intrinsic camera parameters to *ecal*, Willson's program [36] to find the camera translation and rotation. Typically we only need 7 to 10 correspondences to obtain a good calibration of the camera.

Finally, we map the colors from the photograph to the scan (or scans) that it covers. For each 3D point in a scan covered by the photograph, we find its 2D projection on the camera plane, and assign the corresponding pixel color to it. The color of the point is stored together with the point's coordinates in the corresponding octree node. We only store one color sample per point. It would be possible to store multiple color samples per point to support view-dependent effects (such as highlights).

Figure 6 illustrates the imagery registration process. The left image shows a photograph with a few feature points marked by the user, and the right image shows the corresponding 3D points marked on a scan. The right image also shows the estimated camera location and the result of mapping the photograph colors to the scan. The magenta rays go from the camera's position to the true location of the feature points, and the cyan rays go from the camera's position to the expected location of the feature points given the estimated camera parameters. These rays are very close to each other, which indicates an accurate camera calibration.

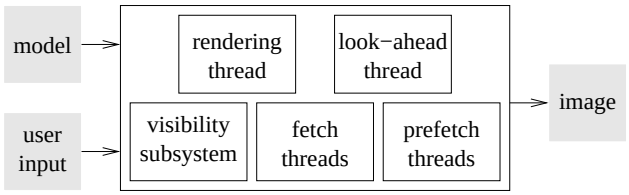


Figure 7: The rendering architecture.

6 Rendering

In the acquisition phase, we capture the geometry and the colors of an environment, and store them in an octree on disk. Our goal now is to walk through the environment at interactive frame rates. Figure 7 illustrates our rendering approach [7]. A rendering thread uses the visibility subsystem to determine the set of octree nodes visible from the user's point of view. For each visible node, the rendering thread sends a fetch request to the fetch threads, which will process the request, and bring the contents of the node from disk into a memory cache. If the cache is full, the least recently used node in the cache is evicted from memory.

To minimize the chance of I/O bursts, which cause abrupt drops in frame rates, there is a look-ahead thread that runs concurrently with the rendering thread. The look-ahead thread tries to predict where the user is going to be in the next few frames, and sends prefetch requests to the prefetch threads. If there are no fetch requests pending, the prefetch threads will bring the requested nodes into memory (up to certain limit per frame based on the disks bandwidth). This speculative prefetching scheme amortizes the bursts of I/O over frames that require little or no I/O, and produces faster and smoother frame rates.

To estimate the set of visible nodes, the visibility subsystem uses the PLP algorithm [16], which is an approximate, from-point visibility algorithm. When we build the octree for an environment, we save the structure of the hierarchy in a separate file. The hierarchy structure (HS) file contains the bounding boxes of the octree nodes and some summary statistics for each node (such as the number of points in the node). The HS file trivially fits in memory. For example, the size of the HS file for a 4-GB octree is 1 MB. The PLP algorithm guesses the visible set based only on the HS file, *without* having to read the geometry inside the nodes. Using PLP, we are able to sustain interactive frame rates (10 frames per second) and high accuracy (above 95% of correct pixels) even for very large environments [7].

The visibility subsystem may also use cPLP [17], a conservative extension of PLP that guarantees 100% accurate images. In conservative mode, however, the visibility subsystem needs to read the exact geometry of each potentially visible node, and the extra number of I/O operations make the frame rate drop.

For interactive rendering, we draw each octree node as an OpenGL vertex array, and we allow the user to set the desired level of detail. If the level of detail is 1, we draw all the points; if it is 1/2, we draw every other point; and so on. We adjust the splat size according to the level of detail using *glPointSize()*. The system can also automatically pick the level of detail based on the user's speed.

For high quality rendering, which requires shading, we find the normal vectors at each point p using MLS projection [2], and choose the normal n that satisfies $p \cdot n < 0$. Because the origin of a scan is at the position $\overline{0}$ of the scanner, and since every scanned point faces the scanner, $p - \overline{0}$ is an estimate of the normal at p. The MLS projection operator is local, i.e., to project a point, we only need to apply the MLS operator over a small neighborhood.

7 Parallel Rendering

When interacting with large, detailed models, it is desirable to visualize these models in high resolution. Our sequential rendering approach only produces low-resolution (1024×768) images at interactive frame rates. We now describe a parallel system that uses our sequential approach as a building block, and delivers 4096×3072-pixel images at the same (or faster) frame rates [8].

Our approach is to use a cluster of PCs to drive a multi-projector display wall. We chose to use a cluster of PCs, as opposed to a high-end parallel machine, for many reasons: a cluster of PCs typically has a better price/performance ratio than a high-end supercomputer; we can upgrade a cluster of PCs much more often than a high-end system, as new inexpensive PC graphics cards become available every 6-12 months; we can easily add or remove machines from the cluster, mix machines of different kinds, and user the cluster for tasks other than rendering; and the aggregate computing, storage, and bandwidth capacity of a PC cluster grows linearly with the number of machines in the cluster [28].

Parallel rendering strategies fall within three main categories, depending on which stage of the rendering pipeline sorting for visible-surface determination takes place [21]. These categories are sort-first, sort-middle, and sort-last. Sort-first approaches divide the 2D screen into disjoint tiles, and assign each region to a different processor, which is responsible for all the rendering in its tile. Sort-middle approaches assign an arbitrary subset of primitives to each geometry processor, and a portion of the screen to each rasterizer. A geometry processor transforms and lights its primitives, and then sends them to the appropriate rasterizers. Sort-last approaches assign an arbitrary subset of the primitives to each renderer. A renderer computes pixel values for its subset, no matter where they fall in the screen, and then transfer theses pixels (color and depth values) to the compositing processors.

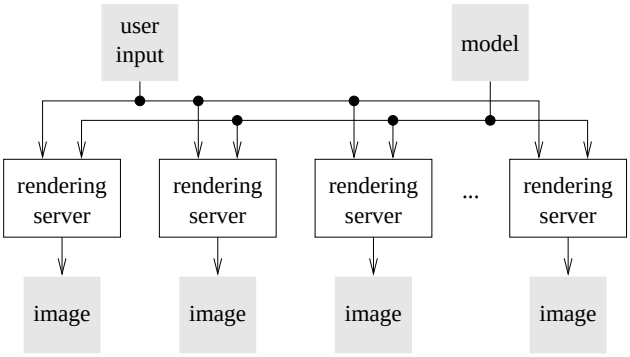


Figure 8: The parallel rendering architecture.

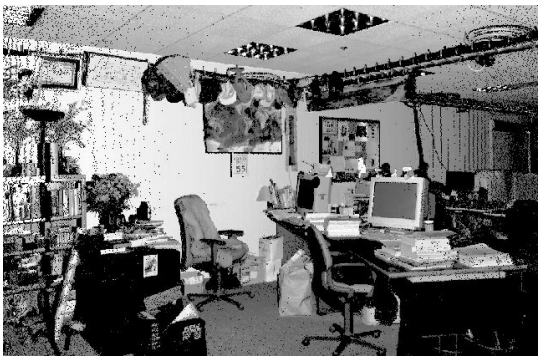
We chose to use a sort-first approach, because sort-first processors implement the entire pipeline for a portion of the screen, which is exactly the case for which PC graphics cards are optimized. A sort-middle approach requires tight integration between the geometry processing and rasterization stages, which is not available in PC graphics cards. A sort-last approach requires high pixel bandwidth, which is also not available in graphics PC cards.

Figure 8 shows our parallel rendering architecture [8]. A client machine is responsible for processing user input. At each frame, the client broadcasts the viewing parameters to the rendering servers. Each rendering server is an MPI task, and runs basically the sequential algorithm we discussed in the previous section, with a few modifications. First, each renderer performs visibility culling using only the view frustum of its display wall tile. Second, each renderer receives inputs events through the network, instead of directly from the user. Finally, we add an MPI barrier at the end of the rendering loop to synchronize the renderers. Each renderer reads the parts of the model it needs from a shared network disk in the file server, and sends the resulting image to one of the display wall projectors. Optionally, each renderer may read its primitives from a local disk.

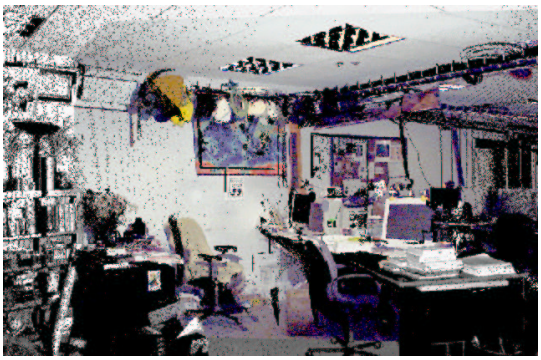
8 Results

Figure 9a shows a photograph of a work space at AT&T's Information Visualization Laboratory. The environment is rich in detail that would be extremely difficult to model by hand. Figure 9b shows part of a 3D scan of that work space. The scan contains about 4 million points, spanning 360 degrees horizontally and 70 degrees vertically. Figure 9c shows the result of mapping the photograph onto the scan. We acquired two other scans of the same room at different positions, and Figure 9d shows the result of merging them. The circle patterns on the ceiling correspond to areas that were missed by one of the scans but covered by another. The total acquisition time was about 1 hour, and the size of the octree containing the three scans was 600 MB.

(a) Photograph



(b) 3D scan



(c) Photograph mapped onto 3D scan

(d) 3 registered scans

Figure 9: AT&T Info Lab.

We tested the performance of the rendering system using a cluster of 16 PCs, and gathered statistics for a pre-recorded 485-frame camera path inside the info-lab model. Each rendering server is a 900 MHz AMD Athlon with 512 MB of main memory, an nVidia GeForce2 graphics card, and an IDE hard disk. The file server has a 400 GB SCSI disk. The client machine is a 700 MHz Pentium III. All machines run Red Hat Linux 7.2, and are connected by a switched gigabit ethernet. Using approximate visibility mode, we were able to render 4096×3072-pixel images of the model, with a mean accuracy of 95.7% of correct pixels, and at a mean frame rate of 9.8 frames per second.

9 Conclusion and Future Work

We have described a complete pipeline for acquiring and rendering point-based models of real-world environments. We have discussed the details of the acquisition process, the limitations of the scanner, our approaches to minimize the acquisition artifacts, and our out-of-core hierarchical representation of the scanned model. We have shown how to efficiently render the model on low-end machines with small memory by using a multi-threaded approach that overlaps rendering, visibility computation, fetching, and speculative prefetching. We have also shown how to build a parallel system that renders high-resolution images on a multi-projector display wall driven by a cluster of PCs.

There are many ways to improve our pipeline. Our geometry registration and imagery mapping procedures are semi-automatic. Also, the geometry registration only takes into account a pair of scans at a time. We are studying approaches that are more automatic, and perform a global alignment [13, 15, 27]. The geometry returned by the scanner may contain holes due to occlusion or because the scanner cannot capture certain surfaces. We are investigating a multi-modal approach that extracts depth information from calibrated photographs to fill in some of those holes. Finally, we would like to accelerate point set surface rendering by exploiting the programmable vertex shaders available in new PC graphics cards such as the nVidia GeForce3.

Acknowledgements

We thank 3rdTech, and in particular Lars Nyland, for help with the scanning hardware. We are grateful to Stanford University and the University of North Carolina at Chapel Hill for providing us with test models. We are indebted to the following people who have given us code, suggestions, and encouragement: Daniel Aliaga, David Dobkin, Thomas Funkhouser, James Klosowski, Jeff Korn, Manuel Oliveira, Emil Praun, Szymon Rusinkiewicz, and Greg Turk. This research was partly funded by CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico), Brazil.

References

[1] 3rdTech. Deltasphere-3000 laser 3D scene digitizer.

[2] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, and C. T. Silva. Point set surfaces. *IEEE Visualization 2001*, pages 21–28, Oct. 2001.

[3] D. Aliaga, J. Cohen, A. Wilson, H. Zhang, C. Erikson, K. Hoff, T. Hudson, W. Stürzlinger, E. Baker, R. Bastos, M. Whitton, F. Brooks, and D. Manocha. MMR: An interactive massive model rendering system using geometric and image-based acceleration. *1999 ACM Symposium on Interactive 3D Graphics*, pages 199–206, 1999.

[4] F. Bernardini, I. Martin, J. Mittleman, H. Rushmeier, and G. Taubin. Building a digital model of Michelangelo's Florentine Pietà. *IEEE Computer Graphics & Applications*, 22(1):59–67, Jan. 2002.

[5] J. F. Blinn. Light reflection functions for simulation of clouds and dusty surfaces. In *Proceedings of SIGGRAPH 82*, pages 21–29, 1992.

[6] J.-Y. Bouguet. Camera calibration toolbox for matlab. http://www.vision.caltech.edu/bouguetj/calib_doc.

[7] W. T. Corrêa, J. T. Klosowski, and C. T. Silva. iWalk: Interactive out-of-core rendering of large models. Technical Report TR-653-02, Princeton University, 2002.

[8] W. T. Corrêa, J. T. Klosowski, and C. T. Silva. Out-of-core sort-first parallel rendering for cluster-based tiled displays. In *Proceedings of the 4th Eurographics Workshop on Parallel Graphics and Visualization.*, 2002. To appear.

[9] C. Csuri, R. Hackathorn, R. Parent, W. E. Carlson, and M. Howard. Towards an interactive high visual complexity animation system. In *Proceedings of SIGGRAPH 79*, pages 289–299, 1979.

[10] S. Fleishman, D. Cohen-Or, and D. Lischinski. Automatic camera placement for image-based modeling. *Computer Graphics Forum*, 19(2):100–110, 2000.

[11] T. A. Funkhouser, C. H. Séquin, and S. J. Teller. Management of large amounts of data in interactive building walkthroughs. *1992 ACM Symposium on Interactive 3D Graphics*, 25(2):11–20, Mar. 1992.

[12] J. Grossman and W. J. Dally. Point sample rendering. In *9th Eurographics Workshop on Rendering*, pages 181–192, Aug. 1998.

[13] O. Hall-Holt and S. Rusinkiewicz. Stripe boundary codes for real-time structured-light range scanning of moving objects. In *Proceedings of the Eighth International Conference on Computer Vision*, pages 359–366, 2001.

[14] B. K. P. Horn. Closed form solution of absolute orientation using unit quaternions. *Journal of the Optical Society A*, 4(4):629–642, Apr. 1987.

[15] D. F. Huber and M. Hebert. Fully automatic registration of multiple 3D data sets. In *IEEE Computer Society Workshop on Computer Vision Beyond the Visible Spectrum (CVBVS 2001)*, Dec. 2001.

[16] J. T. Klosowski and C. T. Silva. The prioritized-layered projection algorithm for visible set estimation. *IEEE Transactions on Visualization and Computer Graphics*, 6(2):108–123, Apr. 2000.

[17] J. T. Klosowski and C. T. Silva. Efficient conservative visibility culling using the prioritized-layered projection algorithm. *IEEE Transactions on Visualization and Computer Graphics*, 7(4):365–379, Oct. 2001.

[18] D. Levin. Mesh-independent surface interpolation. Technical report, Tel-Aviv University, 2000. Available online at http://www.math.tau.ac.il/˜levin.

[19] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk. The digital Michelangelo project: 3D scanning of large statues. In *Proceedings of SIGGRAPH 2000*, pages 131–144, 2000.

[20] M. Levoy and T. Whitted. The use of points as a display primitive. Technical Report TR 85-022, The University of North Carolina at Chapel Hill, Department of Computer Science, 1985.

[21] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32, 1994.

[22] L. Nyland, A. Lastra, D. K. McAllister, V. Popescu, and C. McCue. Capturing, processing and rendering real-world scenes. In *Videometrics and Optical Methods for 3D Shape Measurement, Electronic Imaging 2001, Photonics West*, volume 4309, pages 107–116. SPIE, 2001.

[23] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: Surface elements as rendering primitives. In *Proceedings of SIGGRAPH 2000*, pages 335–342, 2000.

[24] R. Pito. A solution to the next best view problem for automated surface acquisition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(10):1016–1030, 1999.

[25] W. T. Reeves. Particle systems — a technique for modeling a class of fuzzy objects. *ACM Transactions on Graphics*, 2(2):91–108, Apr. 1983.

[26] S. Rusinkiewicz and M. Levoy. QSplat: A multiresolution point rendering system for large meshes. In *Proceedings of SIGGRAPH 2000*, pages 343–352, 2000.

[27] S. Rusinkiewicz and M. Levoy. Efficient variants of the ICP algorithm. In *Proceedings of the Third International Conference on 3D Digital Imaging and Modeling*, pages 145–152, 2001.

[28] R. Samanta, T. Funkhouser, K. Li, and J. P. Singh. Hybrid sort-first and sort-last parallel rendering with a cluster of PCs. In *2000 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 97–108, 2000.

[29] R. Samanta, T. Funkhouser, K. Li, and J. P. Singh. Sort-first parallel rendering with a cluster of PCs. In *Sketches and Applications, SIGGRAPH 2000*, page 260, 2000.

[30] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[31] G. Schaufler and H. W. Jensen. Ray tracing point sampled geometry. In *Eurographics Rendering Workshop Proceedings*, pages 319–328, 2000.

[32] A. R. Smith. Plants, fractals and formal languages. In *Proceedings of SIGGRAPH 84*, pages 1–10, 1984.

[33] S. Teller. Toward urban model acquisition from geolocated images. *Pacific Graphics '98*, Oct. 1998.

[34] S. J. Teller and C. H. Séquin. Visibility preprocessing for interactive walkthroughs. In *Proceedings of SIGGRAPH 91*, pages 61–69, 1991.

[35] R. Y. Tsai. A versatile camera calibration technique for high-accuracy 3D machine vision metrology using off-the-shelf TV cameras and lenses. *IEEE Journal of Robotics and Automation*, RA-3(4):323–344, Aug. 1987.

[36] R. Willson. Freeware implementation of Roger Tsai's camera calibration algorithm. Available online at http://www.cs.cmu.edu/˜rgw/TsaiCode.html.